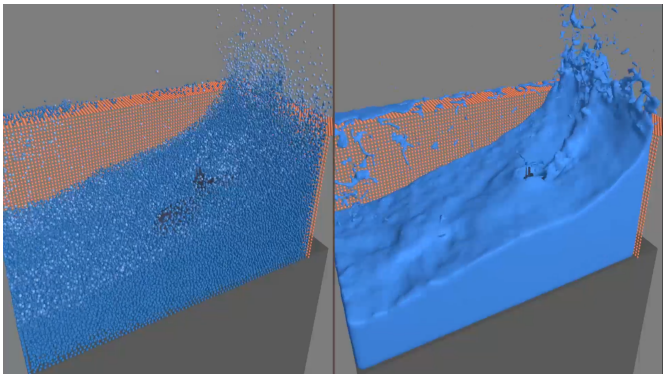
IGR205: IISPH Project Report

Arnault Verret, Gabriel Françon, Anselme Donato

Abstract

The goal of this project was to perform physical simulations of liquids based on a variant of SPH methods called Implicit Incompressible SPH [1] (IISPH). The project was articulated around three tasks: to implement this new SPH model, to generate the mesh representing the fluid surface, and finally to render the resulting animation with physic-based shaders.



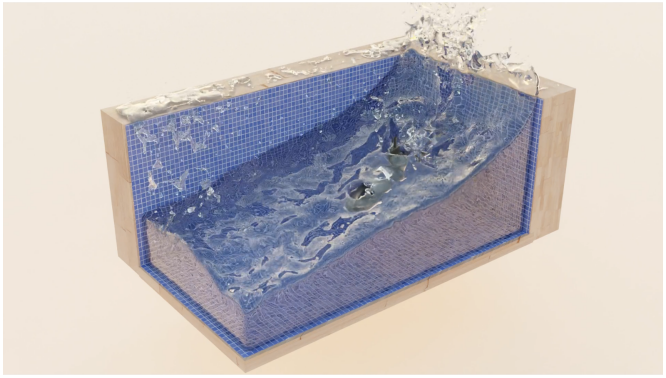


Figure 1: Breaking dam simulation

1. Particle Simulation

The Lagrangian description is a technique to characterize flows, which consists in representing the fluid in the form of particles, and to study their variation with time along their trajectory. In fluid simulation, a very popular approach based on this Lagrangian description is the Smoothed Particle Hydrodynamics (SPH). This method is based on the idea that the physical values associated with a particle over time, like density and pressure, only depend on the state of the particles present nearby.

1.1. SPH Recall

The neighborhood is defined as the set of particles that are spatially located in a predefined radius around the considered particle. However, it would take a long time to verify the Euclidean distance separating each particle from a simulation which generally has tens or hundreds of thousands of particles in motion. In practice, the space is sampled into a grid of cells, each comprising a subset of the fluid particles at a given time of the simulation. It is then very simple to determine which cells are included in the neighborhood of the particle in question, it only remains to iterate on the particles contained in these cells in order to determine if they are indeed included in a radius near the particle.

In fluid mechanics, the trajectory of each particle is described by the Navier-Stokes equations, which are simplified here since the fluid is considered incompressible ($\frac{d\rho}{dt} = 0$)

$\rho \frac{D\nu}{Dt} = \sum_i F_i$ **momentum equation** (conservation of momentum)
$\nabla \cdot v = 0$ **continuity equation** (conservation of mass)

With as forces considered:

$F_{body} = \rho \cdot g$ total of body forces (here only gravity is counted)
$F_{viscosity} = \mu \nabla^2 v$ internal stress forces (viscosity effects)
$F_{pressure} = -\nabla p$ pressure gradient

To solve these equations numerically, it is necessary to discretize them according to time, and then adapt them so that they reflect the state of nearby particles. According to the classic SPH framework (named WCSPH - Weakly Compressible SPH - in the literature), the i-th particle will be governed by the following system :

$\rho_i = \sum_j m_j W_{ij}$
$v_i(t + \Delta t) = v_i(t) + \Delta t \frac{F_{i,body} + F_{i,viscosity} + F_{i,pressure}}{m_i}$

With corresponding forces:

$$F_{i,body} = \rho_i \cdot g$$
$$F_{i,viscosity} = 2\mu m_i \sum_j \frac{m_j}{\rho_j} \frac{\Delta v_{ij} \cdot \Delta x_{ij}}{\|\Delta x_{ij}\|^2 + 0.01h^2} \nabla W_{ij}$$
$$F_{i,pressure} = -m_i \sum_j m_j (\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2}) \nabla W_{ij}$$

(W is a smooth kernel function which weight influence of each neighboring particle according to its distance)

If these equations have the advantage of remaining simple enough to be easily implemented, they present a big weakness: since we approximate the time scheme integration, we introduce errors at each step, which will add up over the iterations. Worse, the error induced grows all the more quickly as a high time step is chosen, which highly limits the general speed of the simulation.

1.2. Incompressibility Error

To limit this error, it is necessary to understand that the main cause of error actually comes from the condition of incompressibility of the fluid. In SPH reasoning, the condition is not verified when:

$$|\sum_{i=1}^{n} (\rho_i / n) - \rho_0| = \epsilon > 0$$

which is what happens a lot with Weakly Compressible SPH. What the group of researchers at the origin of IISPH have imagined is a way to control this error. Their idea is to adopt a predictive/corrective approach for the calculation of the pressure field, described in the following part.

1.3. IISPH Variant

From the general continuity equation $\frac{D\rho}{Dt} = -\rho \nabla \cdot v$, they express the updated density and velocity as a function of the previous ones:

$$v_i(t + \Delta t) = v_i(t) + \Delta t \frac{F_{i,body} + F_{i,viscosity} + F_{i,pressure}}{m_i}$$
$$\rho_i(t + \Delta t) = \rho_i(t) + \underbrace{\Delta t \sum_j m_j v_{ij}(t + \Delta t) \nabla W_{ij}(t)}_{\text{local density deviation, must tend to 0}}$$

They also introduce some "advection" quantities which only rely on non-pressure forces:

$$v_i^{adv} = v_i(t) + \Delta t \frac{F_{i,body}(t) + F_{i,viscosity}(t)}{m_i}$$
$$\rho_i^{adv} = \rho_i(t) + \Delta t \sum_j m_j v_{ij}^{adv} \nabla W_{ij}(t)$$

So to ensure the incompressibility, we thus need to resolve the following system:

$$\Delta t^2 \sum_j \left(m_j \frac{F_{i,pressure}(t)}{m_i} + \frac{F_{j,pressure}(t)}{m_j} \right) \nabla W_{ij}(t) = \rho_0 - \rho_i^{adv} \Leftrightarrow \sum_j a_{ij} p_j(t) = b_i$$

We can deduce from this the general corrective predictive algorithm to apply at each iteration:

1. **Search neighbors:** store the indices j of the neighbors for all particle i;
2. **Predict advection:** compute v_i^{adv} and ρ_i^{adv} for all particle i;
3. **Solve pressure:** determine $\rho_i(t)$ by resolving $\sum_j a_{ij} p_j(t) = \rho_0 - \rho_i^{adv}$ for all particle i;
4. **Correct integration:** deduce $v_i(t + \Delta t) = v_i^{adv} + \Delta t \frac{F_{i,pressure}}{m_i}$ for all particle i.

We will not describe here in detail the technique for solving the PPE, but the strategy adopted in the article and which we have implemented is to use the relaxed Jacobi method. We define a relaxed factor , then the pressure field is iteratively recalculated according to a formula of the form:

$$p_i^{l+1} = (1 + \omega) \cdot p_i^l + \omega \cdot f(\rho(t), p(t), v(t))$$

The corrected density can then be calculated by adapting the formula, and we can therefore re-evaluate the new compressibility error ϵ. By playing on the tolerance threshold η, we therefore have a way to control the compressibility of the fluid [**Algorithm 1**].

Algorithm 1 Pressure solve

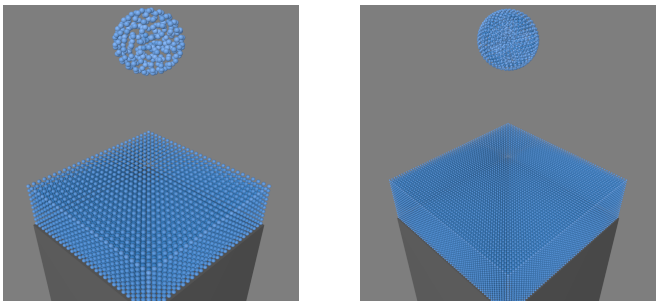
```
while ε > η and l < max_iteration do
    for All particles i do
        Compute p_i^{l+1}
        Deduce ρ_i^{corr}
    end for
    ε = |Σ_{i=1} n(ρ_i^{corr}/n) − ρ_0|
    l ← l + 1
end while
```

Generally, the relaxation factor is evaluated at $\omega = 0.5$, and the compressibility threshold around $\eta = 0.1\%$ to ensure good results. For the initialization of ^0_i, the authors suggest to take $p_i^0 = 0.5 p_i(t - \Delta t)$ according to some experimental tests they carried out.

1.4. Advantages of IISPH over Classic Approach

By limiting the compressibility of the fluid, the pressure field obtained is more plausible, and the simulation is therefore more effective, coming much closer to physical reality. This is particularly the case for the behavior at the limits and in configurations of great agitation of the fluid

In addition to gaining in accuracy, the described model also allows to make the integration scheme much more robust over time, thus allowing to integrate over larger time steps and thus to make the simulation more efficient. For WCPSH, we had to limit ourselves to 0.5 ms, whereas here we can easily integrate over 5 to 20 ms.



h=0.5 h=0.25

Figure 2: Initialization of a fluid drop and a basin with different particle spacing h

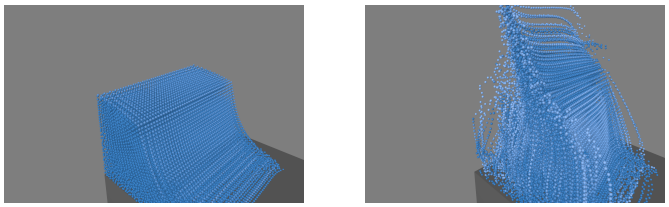


Figure 3: Good and bad sampling of the fluid mass in a breaking dam scenario, resulting in a stable or unstable simulation

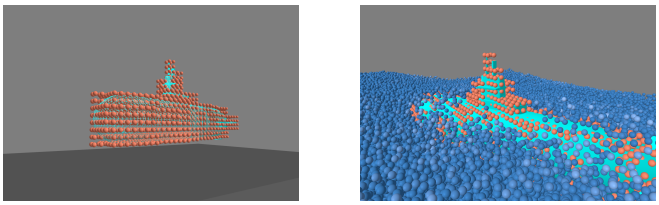


Figure 4: A submarine mesh sampled with boundary particles after executing the rasterization process

1.5. Handling Boundaries

Boundary management in SPH frameworks is the subject of much work. The advantage of the IISPH model is that it can be easily integrated in any existing framework since it only modifies the calculation of the pressure field. In our case and in order not to make the solver more complex, we adopt a classic approach by adding static boundary particles which will contribute to the fluid density at the boundaries.

At the beginning of the simulation it is necessary to calculate the local density number $\delta_b = \sum_j W_{bj}$ of each boundary particle, which roughly corresponds to the local "wall density". Each neighbor boundary particle will then contribute to fluid calculation at a rate of $\psi_i = \rho_0/\delta_b$. (Same formula as before, except that the neighbor sum is weighted by m_i for fluid neighbors, and by ψ_i for boundary ones).

1.6. Sampling the Scene

When initializing a scene containing particles, it is essential to ensure that the initial density is consistent with the rest density of the fluid. If the particles are too far apart, then the incompressibility condition is not verified. If on the contrary the particles are too close together, the fluid will be in an overpressure situation and the particles will explode in all directions, repelling each other (figure[3]).

The correct sampling is obtained when the particles forming the fluid mass are spaced from each other and with boundaries by a constant particle spacing h, corresponding to the particle diameter. The mass of each particle is then equal to $m_i = \rho_0 \cdot h^3$ which means the overall density of the fluid will be equal to ρ_0. This parameter h is decisive, since it allows in fact to control the resolution of the fluid (figure[2]). By decreasing it, you will simulate more precisely the behavior of the fluid, but also increase memory usage and computing time.

The spacing regularity rule also applies to boundary particles. For simple boundaries like the bounding walls, this is not a big deal. However, we may need to integrate more complex meshes in our scene, as in our scenario of breaking dam where a submarine mesh is put in the way of the fluid. We thus need an algorithm able to convert any set of triangles into a set of particles.

The first idea was just to put a particle at each vertex, but for stretched triangles it doesn't work so we came up with another approach which is to rasterize the triangles in a 3D space grid. The main idea is to take each triangle one by one, and for each cell of the grid check if the triangle intersects (figure[4]). We optimize the algorithm by creating a bounding box around the triangle to check less cells. To check for the intersection, we use the Fast 3D Triangle-Box Overlap Testing [2].

If this gives a lot more particles, slowing down the simulation, the simulation works much better and we don't see particles going through the object. It must be said that the best approach is still to do it analytically, and most primal shapes can be put into equations to get the best possible particle positions.

1.7. Multi-Threading using the OpenMP API

During an iteration, a large number of for loops are performed on all the fluid particles, and each calculation inside the loop is independent of the one performed on the other particles. This configuration is thus very well adapted to execute each iteration of the loop in parallel threads. Beware however, it is not possible to entrust the calculation of a quantity of every particle to each thread (for instance, thread 1 searches neighbors, thread 2 calculates the density, thread 3 the pressure, ...) since we need to know the quantities of the neighbors of each particle to calculate the value of the next quantity.

To do this, we use the openMP API which allows in just one line to specify to the compiler to distribute the

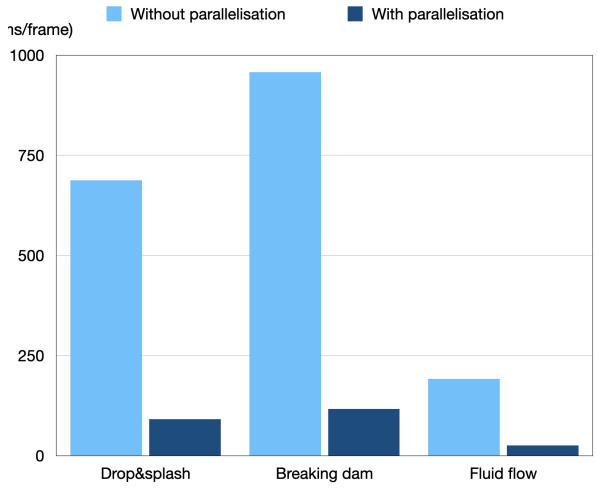


Figure 5: Computation time saved when using multi-threading in SPH simulation

iterations of a loop to all the available threads of the CPU [**Algorithm 2**].

Algorithm 2 Density Multi-Threading

```
#pragma omg parallel for
for (int i=0; i < _fluidCount; i++) do
    computeDensity(i);
end for
```

This simple but powerful addition can drastically increase the speed of solving fluid equations, especially when using a processor with many cores (figures[5]).

Processor: Ryzen 7 5800X (8 cores, 16 threads)
Graphic Card: NVIDIA GeForce RTX3070
RAM: 16Go 3200MHz

2. Surface Reconstruction

The IISPH model allowed us to simulate fluids in the form of particles, but it does not give us the means to really visualize this fluid in the form of a single deformable 3D element. To do this, it is necessary to reconstruct the surface of this fluid, based on the distribution of the particles at each frame.

2.1. Isosurface and Marching cubes

To fully understand the process of surface reconstruction of the liquid, it is necessary to introduce the notion of isosurface. An isosurface is basically a set of points in a delimited space where a given scalar function is constant. It is therefore the 3D analogue of an isoline. For example, for a sphere or radius r_0 centered in x_0 and given the following function $f(x) = |x - x_0|$, the isosurface can be described as : $\{x \in \mathbb{R}^3 : f(x) = r_0\}$.

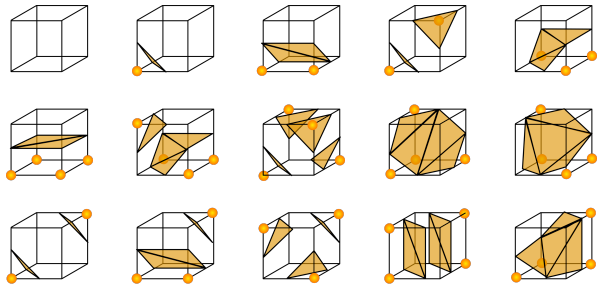


Figure 6: Some configurations of triangles generated inside a cell, depending on how the isosurface crosses it. (highlighted vertices inside the fluid)

A popular algorithm to construct the polygonal model of an isosurface from a given 3D scalar field is the Marching Cubes algorithm. Although first published in 1987, the algorithm remains widely used because it combines simplicity with high speed, since it works almost entirely on lookup tables.

To create the mesh of an isosurface, the algorithm is based on a 3D rectangular grid which discretizes the space. Given a cell defined by its scalar values at each of its 8 nodes, it is necessary to create planar facets that best represent the isosurface through that grid cell. The isosurface can intersect this cell in several ways, each possibility being characterized by the number of nodes that have values above or below the isovalue (figure[6]).

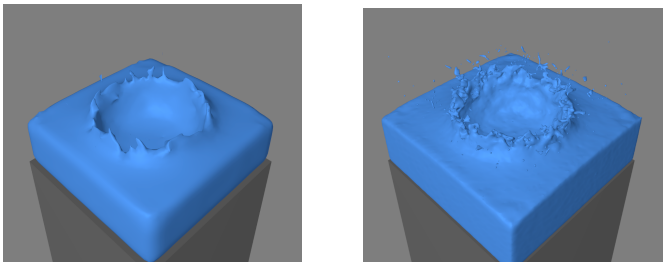
For example, if a vertex is above the isosurface and an adjacent vertex is below the isosurface, we know that the isosurface intersects the edge between those two vertices. The position at which it intersects the edge is then determined by linear interpolation. By iterating over all the cells of the grid, we generate the triangles that will make up the surface.

We haven't personally implemented the Marching Cubes algorithm, as there are already plenty of them free to use. We have thus recovered the version produced by Paul Bourke, which he describes very well on his website in the article Polygonising a scalar field [3].

2.2. Signed Distance Field

The question that remains is what scalar field and isovalue to define in the case of SPH simulation. A 2005 article dealing with Animating sand as a fluid [4], proposes a scalar field $\phi(x)$ representing the signed distance to the real surface of the fluid. So when $\phi(x) < 0$, it means x is below the surface i.e inside the fluid, and when $\phi(x) > 0$, x is above the surface i.e outside the fluid. The isovalue which therefore defines the surface of the fluid is 0.

To assess such a field, the researchers decided to start from the distance function of a single spherical particle, that we had already introduced previously. Adapted to isovalue 0, we can redefine the surface of a single spherical



cell size of 2h cell size of h/2

Figure 7: The surface better wraps the particles around when increasing the resolution of the grid, giving better details on the splash effect

particle by the following distance field: $\phi_{singleparticle}(x) = |x - x_0| - r_0$

We can then adapt this expression to include multiple particles in the neighborhood of x, and weight their contribution according to a well suited smooth kernel:

$$\phi(x) = |\sum_i w_i x_i - x| - r_0 \text{ where } w_i = \frac{k(|x-x_i|/R)}{\sum_j k(|x-x_i|/R)}$$

In our code, we chose $k = max(0, (1 - s^2)^3)$ as smooth kernel, and R is the radius of the neighbors area, generally equal to twice the particle spacing : $(R = 2h$. The result field can be interpreted as an estimation of how far a point is from the surface.

2.3. Grid resolution

It only remains to evaluate this field at each of the nodes composing the reconstruction grid, and to apply the marching cubes algorithm to generate a mesh of the fluid isosurface. The parameter that must be specified, however, is the resolution of this grid. Just like the particle simulation grid, reducing the size of the cells allows to better wrap the particles around and thus embrace the moving fluid more faithfully (figure[7]), but this comes at the cost of memory usage and computation time : dividing the resolution by two is equivalent to dividing each cell by 8 and thus multiplying by 8 the number of cells to store and evaluate. Best compromise seems to be a cell size of $h/2$, i.e a resolution increased by 4 compared to the SPH grid.

2.4. Mesh smoothing

The mesh obtained is already a faithful reconstruction of the surface induced by the particle distribution, but it can be noticed that it is sometimes irregular and does not really look like the smooth surface characteristic of fluids. Indeed, the Marching Cubes algorithm tends to introduce some roughness during the creation of the triangles, resulting in this unwanted noise effect.

To correct this aspect, there are complex approaches seeking to correct the simulation of the particles using anisotropic smooth kernels, which gives very good results but considerably increases the temporal complexity of the

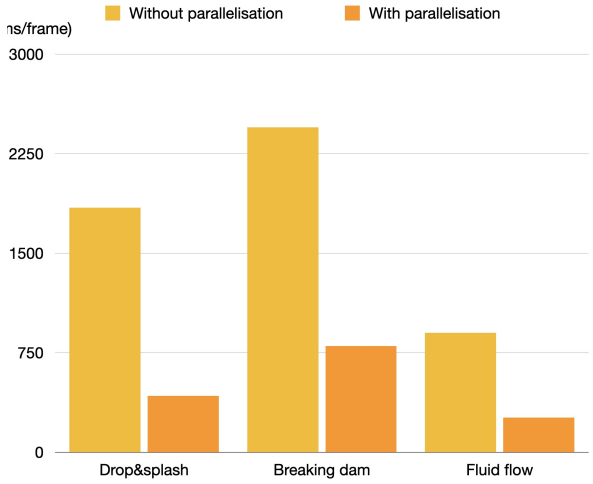


Figure 8: Computation time saved when using multi-threading in Surface reconstruction

simulation. A cheaper but still relatively efficient approach is to apply a simple smoothing algorithm to the mesh, as a post-processing. In our case, we use Laplacian smoothing[5]. The general idea is to recalculate the position of each vertex according to the weighted sum of the positions of adjacent vertices. The smoothing coefficient can then be controlled by adapting the number of iterations of the algorithm.

However, it is important to note that the operation tends to reduce the overall size of the mesh, and in particular the isolated water drops. There is also a risk that the smoothing in one frame is not consistent with the one obtained in the previous frame, making the animation of the fluid strange, which is why we can't choose t high the coefficient. After various tests we decide that 3 smoothing passes is a good compromise between smoothness and consistency.

2.5. Parallelisation and other optimization

We have seen before that a lot of the calculation can be done in parallel on the CPU using OpenMP. This should also be the case for the surface reconstruction as, in theory, each cell is processed independently. We therefore compute the signed distance field with multi-threading (figure[8]). However, we here face a major difficulty of multi-threading which is the memory allocation. During the marching cubes algorithm, each new triangle is stored into one dynamic array. As we allocate one thread for each cell (or by batches), two threads could potentially try to access the array at the same time, which causes crashes. This problem can be fixed using lockers, but we didn't want to dig into fences and semaphores. That is why the slowest part of our algorithm is the surface reconstruction.

We could mention that if we mostly use the CPU, a common practice is to transfer the computation part to the GPU using compute shaders (or Cuda). The main trouble

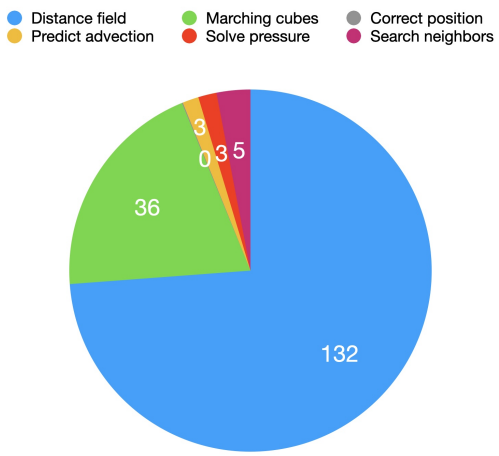


Figure 9: Repartition of average time execution across the Breaking dam simulation (in ms)

we faced here was the communication between the CPU and the GPU where, like the multi-threading, we should have used fences and semaphores to "wait" for the GPU to compute before going back to the CPU. But according to our results CPU based, one could expect the simulation to be almost real-time with a GPU based simulation for less than 200k particles.

3. Rendering

3.1. Simulation Considered

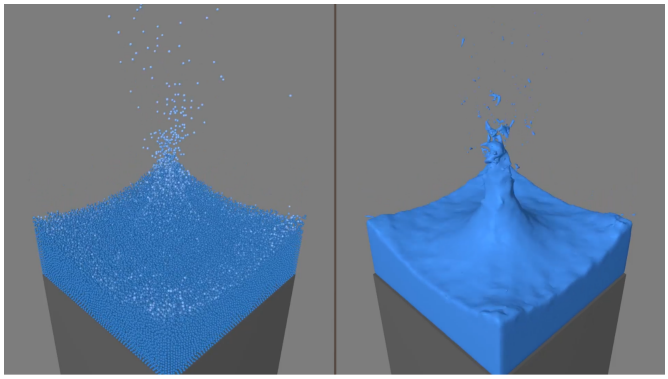
Breaking dam - A classical dam failure simulation, including a complex solid object, here a mini submarine, to observe collisions of fluid with static objects (figure[1]).

Drop and splash - A simulation where a spherical volume of water, similar to a drop, is released over a water basin, in order to observe the splash and water crown effects (figure[10]).

Fluid flow - A simulation that presents a flow of liquid between two containers of different shape. It allows testing the robustness of the model to continuous fluid flows (figure[11]).

3.2. Rendering Using Blender

In order to do the rendering in Blender, we used a stop-motion approach : once we have generated one mesh per time step in Vulkan, we save them all as a sequence of .OBJ files, and then import the sequence in Blender as an animation using an add-on called stop-motion-OBJ [6]. Basically what the add-on does is load and then unload each mesh in memory when playing the animation, letting you choose the size of the memory cache (prevent saturating the RAM memory).



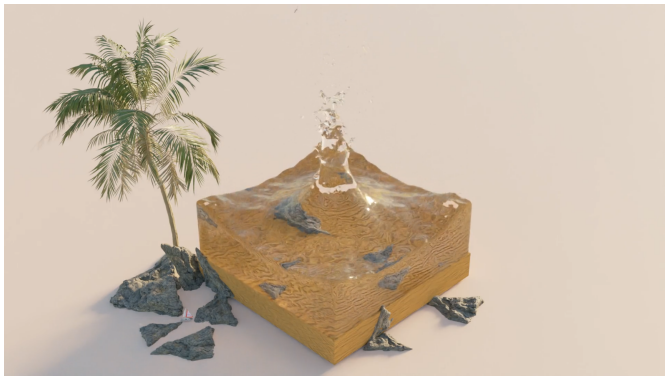
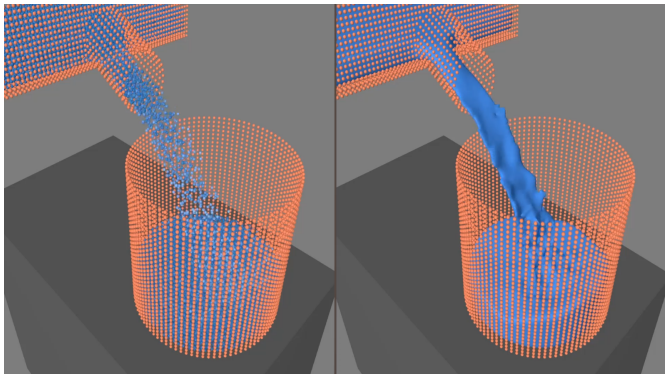


Figure 10: Drop and splash simulation - incorporated into a Blender composition for more visually appealing results.



wine material

whisky material

Figure 11: Fluid flow simulation - using different base color and volume shaders to shade various liquids

The next step is to use Blender in order to make that mesh look like an actual fluid in the render. For that, we used a simple liquid material, mainly based on three things:

- We put the roughness at 0% and the transmission at 100% to make it clear and transparent
- We used the same IOR (Index Of Refraction) as water : 1,33
- For the simulations were the liquid is colored, we added some volume absorption for the color too

With only those few simple settings we were able to achieve a pretty convincing look for our fluid. We then added a skybox and a scene with a few elements to add reflections and distortions in the fluid, which help to see its motion more easily. This part required some tweaking to make sure that everything looked nice in the end. We were especially concerned about some weird reflections on the edge of the fluid that seemed unnatural, or that were making it hard to see the inside. We get rid of the first one by making the liquid slightly overlap with the scene, to ensure that there were no gaps between them where the light could reflect in unpredicted ways. For the second, we tested different values for the skybox (sun intensity and inclination, composition of the "atmosphere" in Blender) until we were satisfied with the result.

At that point everything is clear for the final render. As previously mentioned, we used a stop-motion approach which means that we rendered one image per mesh in Blender, stored every image in a folder and then later on combined every image into a GIF or a video. We used Blender's physic-based engine (Cycles) to get realistic lightning effects (diffraction in the liquid thanks to ray-tracing, essential to render realistic transparent liquid). As expected, this resulted in nicer results but longer rendering times so we had to make some compromises. We limited the number of light bounces, and more importantly drastically reduced the number of samples, and taking advantage of Blender's good denoiser to still get a crisp image at the end.

4. Final thoughts

Through this project, we succeeded in implementing a more efficient and effective SPH simulation model than the classical approach studied earlier in the year, and managed to exploit the Marching Cubes algorithm to reconstruct the fluid surface mesh. We also optimized our program by taking advantage of the multi-core structure of modern CPUs, and made some photorealistic renderings in Blender to better appreciate the results obtained.

There are many ways to continue our work: better optimization of the isosurface generation, implementation of an adaptive particle spacing system to better capture liquid motions around complex boundaries without increasing computational time, complementing our physical model with other interactions like surface tension forces, etc.

References

[1] Markus Ihmsen & al. March, **Implicit Incompressible SPH** (2014).
https://cg.informatik.uni-freiburg.de/publications/2013_TVCG_IISPH.pdf

[2] Tomas Akenine-Möller, **Fast 3D Triangle-Box Overlap Testing** (2001).
https://fileadmin.cs.lth.se/cs/Personal/Tomas_Akenine-Moller/code/tribox_tam.pdf

[3] Paul Bourke, **Polygonising a scalar field** (1994).
http://paulbourke.net/geometry/polygonise/

[4] Yongning Zhu & al., **Animating sand as a Fluid** (2005).
https://www.cs.ubc.ca/~rbridson/docs/zhu-siggraph05-sandfluid.pdf

[5] David A. Field, **Laplacian smoothing and Delaunay triangulation** (1988).
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.648.3296&rep=rep1type=pdf

[6] **Blender add-on stop-motion-OBJ**.
https://github.com/neverhood311/Stop-motion-OBJ